(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 895 148 A1

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
03.02.1999 Bulletin 1999/05

(51) Int. Cl.[6]: **G06F 1/00**

(21) Application number: **97113262.6**

(22) Date of filing: **31.07.1997**

(84) Designated Contracting States:
AT BE CH DE DK ES FI FR GB GR IE IT LI LU MC NL PT SE

(71) Applicant:
SIEMENS AKTIENGESELLSCHAFT
80333 München (DE)

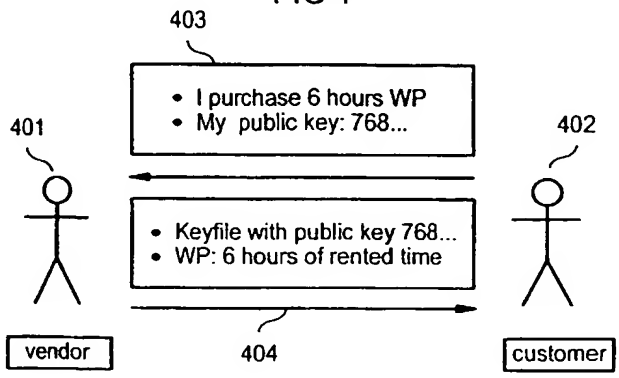
(72) Inventor: **Benson, Glenn, Dr.**
81739 München (DE)

(54) Software rental system and method for renting software

(57) A software rental system is provided comprising at least one rented program permitting at least one service to a customer with a customer's reponse means, wherein

- said rented program has no access to a customer's private keying material,
- using asymmetric cryptography, said customer's reponse means proves to the rented program, that said customer's reponse means has access to the customer's private keying material, and
- said rented program does not permit said at least one service to said customer unless the proof is successful.

FIG 4



EP 0 895 148 A1

Description

[0001] With few exceptions, most computer programs are instantiations of intellectual property and execute upon demand after installation. Normally, one can build and install an inexpensive copy of a computer program by little more than simply executing a copy command. A physical asset such as an automobile, on the other hand, cannot be easily copied. As a result, it is much easier to provide rental of physical assets than rental of computer programs. In the case of a physical asset, the renter first pays a rental fee, and then physically takes possession of the asset. At the conclusion of the rental period, the renter returns the physical asset to the owner. In the case of software, on the other hand, it makes little sense for the customer to return the program to the owner because one cannot guarantee that the customer refrained from sequestering his or her own backup copy. In the absence of adequate security measures, a customer acting in the role of an attacker, could potentially rent the software for a short period of time and subsequently use the sequestered backup without paying further rental fees.

[0002] In this invention we define software rental as a computer system and method that securely stores rental (usage) records. For example, consider the time-of-use rental metric. If the customer, executes the rented software for one hour on the first day and two hours on the second, then the secured audit trails show one hour at the end of the first day and three hours at the end of the second day. Secure software rental implies that a customer cannot defeat system security by purging, replacing, or modifying audit trails. Normally, the software continually monitors the audit trails to determine when a threshold is exceeded. So, if the example software has a five hour threshold, then the customer may execute the software for two more hours and then the software stops. Another example threshold is the total amount of times that the software may execute.

[0003] Some rental mechanisms that have all the properties listed above of our invention currently exist, e.g., Dongles [1]. Dongles have non-volatile memory which may be protected by passwords. This password protected memory may potentially used for software rental. A characteristic of this rental mechanism is that it requires the assistance of a secured rental device, e.g., a Dongle. The secured rental device contains Secured Updateable Storage Locations (SUSLs) that record information related to usage of the rented software. Each SUSL has the property that the SUSL resides on a secured device and provides protection against attack. Normally, at least one SUSL for each unit of rented software is required. For example, if a customer rents a word processor, a spread sheet, and a game, then the rental device(s) must provide at least three SUSLs. These SUSLs are relatively expensive and difficult to administer, when compared with other storage on the customer's machine, e.g., memory or disk space.

[0004] Software rental, furthermore, significantly differs from a subscription to a network service. For example, suppose a software vendor provides a server to which customers connect via their software clients. During the period of the connection, the server audits usage records, e.g., connect time. The vendor assesses charges based upon the information recorded in the server's audit trail. This client-server example differs from this present invention because we do not necessarily require an on-line presence by the software vendor. Rather, after obtaining permission to use the rented software, the customer executes the software without any required network connections. Furthermore, the subscription service does not prevent the customer from caching frequently used items.

[0005] An overview on asymmetric cryptography, for example on the RSA scheme, and probabilistic encryption, for example the Blum-Goldwasser probabilistic public-key encryption scheme can be found in [2].

[0006] An overview over different probabilistic proof schemes, for example zero knowledge proof schemes (e.g. Feige-Fiat-Shamir scheme, Guillou-Quisquater scheme, Blum-Feldmann-Micali scheme, Brassard scheme, Crepau scheme, etc.) or witness hiding proof schemes (e.g. Feige-Shamir scheme, etc.) can be found in [2].

[0007] An overview of digital signature schemes (e.g. Rivest-Shamir-Adleman, etc.,) and a formal mathematical definition of digital signatures can be found in [2].

[0008] An example of a message digest function (otherwise known as a one-way hash function) is MD5 [3]. It is computationally infeasible or very difficult to compute the inverse of a message digest.

[0009] In [4], cryptographic randomness from air turbulence in disk drives is described.

[0010] The Chi-Square Test, the Kolmogorov-Smirnov Test, and the Serial Correlation Test are described in [5].

[0011] An asymmetric cryptographic mechanism includes public keying material and corresponding private keying material. It is computationally infeasible to compute the private keying material when given no more information than the corresponding public keying material. In this invention, we use asymmetric cryptography in interactions between two parties, A and B. A proves to B that A has access to private keying material and B validates the proof. A does not disclose the private keying material to B.

Digital signature scheme

[0012] A digital signature is an electronic analog of a handwritten signature. A digital signature proof involves at least two parties, A and B. After posting his or her public keying material to a public location, A encrypts a message using the private keying material. Since anyone may access the public keying material, there is no message secrecy. However,

since A is the only customer with access to the private keying material, no one else can "forge A's signature" by performing the encryption. Anyone may validate A's signature using the public keying material - simply decrypt using A's public keying material.

[0013] An asymmetric confidentiality proof involves at least two parties, A and B. A possesses private keying material and B has no access to A's private keying material unless B discloses the private keying material itself (which B should not do). At the beginning, A and B have no shared secret. During the method, a shared secret becomes known to A and B.

[0014] In all asymmetric cryptographic schemes, each customer may post his or her public keying material to a publicly accessed directory without compromising the corresponding private keying material. The customer usually should guard his or her private keying material as a close secret; otherwise, the cryptographic system may not guarantee correctness (secrecy). The best known mechanism for protecting one's private keying material is through the use of a smart card. In this case, the smart card is a device with no interface for releasing private keying material (in a non-cryptographically protected form). All cryptographic operations that directly reference the private keying material are performed on the smart card itself. As a result, no one can discover the contents of the private keying material stored on a smart card.

[0015] Although smart cards provide the best protection, social factors of electronic commerce may provide a role in ensuring private keying material protection. One of the significant difficulties associated with asymmetric encryption services is authentication. For example, if A posts his or her public keying material to a public directory, then how does B assess validity? That is, a pirate may attempt to masquerade as A but post the pirate's keying material. Some commercial organizations provide solutions to this problem by acting as Certification Authorities (CA). For (possibly) a fee, the CA solicits identifying material from potential customers such as a driver's license or passport. After validating the identifying material, the CA posts the customer's public keying material to a public directory, and the CA signs a certificate (using a digital signature with the CA's private key) that holds the customer's public keying material. Standardized services, for example X.500, may be adopted to help facilitate the use of directories that contain public keying material.

[0016] Once a customer posts his or her public keying material to the CA, the customer will probably make an extensive effort to protect his or her private keying material. For some asymmetric keys, if the customer's private keying material were to become unknowingly compromised, then the customer would have cause for significant concern. For example, in the case of RSA keys that can also be used for digital signatures, networked vendors could potentially authorize electronic commerce transactions.

[0017] The object of the present invention is to create a cryptographically secure software rental system.

Summary of the Invention

[0018] According to the invention there is provided a software rental system comprising at least one rented program permitting at least one service to a customer with a customer's response means, wherein

- said rented program has no access to a customer's private keying material,
- using asymmetric cryptography, said customer's response means proves to the rented program, that said customer's response means has access to the customer's private keying material, and
- said rented program does not permit said at least one service to said customer unless the proof is successful.

[0019] According to a further aspect of the invention there is provided a software rental system comprising at least one rented program and private keying material such that the software rental system securely stores audit trails, wherein the correctness of said audit trails is validated using asymmetric cryptography.

[0020] According to a further aspect of the invention there is provided a method of distributing a program to a plurality of customers wherein each customer has a software rental system as described above, and wherein every customer receives an identical copy of said program.

[0021] According to a further aspect of the invention there is provided a method for renting software comprising at least one rented program permitting at least one service to a customer with a customer's reponse means, wherein

- said rented program has no access to a customer's private keying material,
- using asymmetric cryptography, said customer's reponse means proves to the rented program, that said customer's reponse means has access to the customer's private keying material, and
- said rented program does not permit said at least one service to said customer unless the proof is successful.

[0022] According to a further aspect of the invention there is provided a method for renting software comprising at least one rented program and private keying material such that the software rental system securely stores audit trails, wherein the correctness of said audit trails is validated using asymmetric cryptography.

[0023] We provide a new mechanism in which the number of required SUSLs is not a function of the number of rented programs. In particular, this invention requires only a single SUSL regardless of the number of rented programs or whether or not these programs simultaneously execute. For example, a customer could potentially simultaneously execute 100 or more rented programs while using only a single SUSL. The SUSL requires no more than 128-bits in order to protect an unlimited number of rented programs. In some cases a single 32-bit SUSL or a single 64-bit SUSL may be sufficient. As we will describe later, we use the SUSL to denote a single non-negative integer. We use this integer as a counter that begins at an initial value (normally zero) and ends at the highest value that can be represented by the counter. Under normal circumstances, it is not possible to count through all of the possible numbers that can be represented by a 128-bit counter. Even if one were to build a machine that could increment the counter trillions of times each second (one would not normally want to count this fast), then it would take millions of years to reach the maximum value of the counter ($2^{128} - 1$).

[0024] This invention also provides a new method for distributing rented software. In this method, a customer may rent new applications, from any vendor whenever he or she wishes. The customer does not need to install new external rental devices or new SUSLs.

[0025] The main motivation from the perspective of security for requiring at least one SUSL is to protect against the *backup and restore attack* as described below. Suppose that a rented application requires no SUSL. An attacker may break security by performing the following steps. First, the attacker in the role of an authorized rental customer, legitimately rents a program. Next, the attacker builds a full system backup of all of the non-volatile memory on the attacker's machine. Next, the attacker executes the rented program. Next, the attacker shuts down and reboots his or her machine. Finally, the attacker restores the state of the non-volatile memory to the original state at the time of the backup. At this point, the attacker has destroyed any possible record that the program was previously rented. An SUSL, on the other hand, counters this attack because an SUSL has the property that the information contained in the SUSL cannot be modified in an unauthorized manner. As a result, the attacker's restore operation cannot overwrite the SUSL with the purpose of defeating security.

[0026] Since we cannot ensure that a customer deletes all copies of the software at the conclusion of the rental period, we provide an alternate approach. We lock the software so that the customer cannot execute the software without first providing an appropriate key. At the end of the rental period, we prohibit the customer from subsequently using this key or any copy of the key. Therefore, at the conclusion of the rental period the customer can no longer use the software because the customer will no longer be able to provide an appropriate key.

[0027] In a further aspect of the invention there are multiple rented programs, wherein

- said rented programs access the same public keying material, and
- customer's private keying material is stored on a smart card.

[0028] To further improve the security it may be advantageous that

- at least one usage parameter of said at least one program is stored in an audit trail, and
- said rented program does not permit said at least one service to said customer unless said at least one usage parameter is within a predetermined threshold.

[0029] In a further aspect of the invention said at least one usage parameter is a rental time period for renting said at least one program.

[0030] In a further aspect of the invention said at least one usage parameter is an amount of usage of said at least one program.

[0031] To further improve the security it may be advantageous that

- said at least one usage parameter is checked multiple times, and
- said rented program does not permit said at least one service to said customer unless said at least one usage parameter is in a predetermined interval each time it is checked.

[0032] To further improve the security it may be advantageous that there is provided a rental server that synchronizes access to a smart card on which a customer's private keying material is stored and/or to an audit trail in which usage parameters of said at least one program is stored. Thus, there is a central instance which is responsible for much of the software rental (but not necessarily security).

[0033] In a further aspect of the invention said program stores said at least one usage parameter in said audit trail.

[0034] In a further aspect of the invention, aspects of said audit trail are stored securely.

[0035] In a further aspect of the invention, portions of said audit trail are stored on an medium that is not secure. This makes the system easier and less expensive to be realized without sacrificing security. However, aspects of said audit

trail are stored in an SUSL (the SUSL stores a single counter value).

[0036] To further improve the security it may be advantageous that the smart card performs an atomic routine that both executes a digital signature and an operation computed over an SUSL of said smart card, where the value held in the storage location changes over time.

[0037] To further improve the security it may be advantageous that the system includes a keyfile for holding public keying material.

[0038] To further improve the security it may be advantageous that said public keying material held in said keyfile is cryptographically secured, whereby it is computationally not feasible to alter any portion of the keyfile, including the first public keying material, without altering the challenge means (defined later).

[0039] To further improve the security it may be advantageous that keyfile includes information identifying said customer said program has been supplied.

[0040] To further improve the security it may be advantageous that said keyfile includes decoy bits for disguising the said public keying material held therein.

[0041] To further improve flexibility, it may be possible to use software rental without defining a threshold. In this case, the software may be rented without limit. However, the customer is trusted to periodically send his or her audit trails to the vendor and submit associated payment.

Brief Description of the Drawings

[0042]

Figure 1 is a block diagram showing the architecture of the software rental system.

Figure 2 is a block diagram showing the software components that are required to be installed in the customer's machine.

Figure 3 is a flowchart showing the operation of a random number generator used to generate nonces.

Figure 4 is a block diagram showing a rental software purchase scenario.

Figure 5 is a block diagram showing the smart card architecture.

Figure 6 is a block diagram showing an example audit trail.

Description of an Embodiment of the Invention

[0043] One embodiment in accordance with the invention will now be described by way of example with reference to the accompanying drawings.

[0044] Figure 1 illustrates the system's 100 architecture. Potentially multiple applications (programs) of software reside on the system where each application has its own keyfile, which is explained in detail later. Figure 1 illustrates three applications, a Word Processor 104, a Spread Sheet 105, and another application 106 which access keyfiles 101, 102, and 103, respectively. In some cases multiple applications 104, 105, 106 may share a common keyfile.

[0045] Each of the applications 104, 105, 106 accesses its keyfile 101, 102, and 103 to extract the customer's public keying material as described later in detail.

[0046] Each application vendor inserts rental instructions into a copy protected program. These rental instructions create log records, e.g., 109, 110, 111. For example, every fifteen minutes that Word Processor 104 executes, Word Processor 104 creates the following log record: „Word Process WP with public key 9828a8c12a5873654bac684517d3afe3 executed for 15 minutes" (note that the record could store the message digest of the public keying material rather than the public keying material itself). Next, the application sends its log record to a Rental Server 107. The Rental Server 107 inserts the log record at the end of a secure audit trail 108 stored at a potentially unsecured storage location, e.g., a file on a disk. The Rental Server 107 relies on the assistance of a Smart Card 112 for security.

[0047] An application, e.g., 104, 105, or 106, may choose to create a log record that contains any arbitrary string of bits with arbitrary length. In addition, or in lieu of recording time, an application could potentially log the number of times that the application or some of its modules executes. For example, SS 105 could potentially append a single log record each time SS boots: „Application SS with public key 768230aac8239d9df88cfe3c7b832a is executing". Different types of audit records, e.g., time of usage or number of times that usage occurred may appear in the same audit trail. Multiple rented applications may simultaneously use the same audit trail.

[0048] One obtains software rental by matching thresholds against the audit trail. Figure 4 illustrates a customer 402, who rents software from a vendor 401. First, the customer 402 sends a request to rent the software to the vendor 401 in an order request 403. In this example, the customer purchases six hours of application 104. After receiving payment, the vendor sends to the customer a keyfile 404 that contains a usage authorization. In this case, the keyfile 404 permits six hours of execution by application 104. The later described keyfile 404 may potentially contain other information, e.g., copy protection or licensing information.

[0049] Periodically, the rented application, e.g., word processor (application) 104, examines the audit trail 108. If the audit trail 108 is not valid, then word processor 104 does not permit itself to be rented. However, if the audit trail 108 is valid, then the application 104 analyzes the audit trail and compares the analysis against the keyfile 404. For example, application 104 counts the number of log records that describe 15 minute intervals. Next, application 104 looks into the keyfile 404 to locate a rental threshold which in this present example is 6 hours (24 x 15 minute intervals). If application 104 locates fewer than 24 of its log records denoting 15 minute intervals, then application 104 continues executing. Otherwise, application 104 does not permit itself to be rented. In the latter case, the customer must purchase a new keyfile in order to continue renting application 104. If application 104 were to exceed its rental threshold, then other applications, e.g., spread sheet 105 and other application 106 would not be effected. That is, each rented application views its own records from the audit trail without interpreting records created by other applications.

[0050] From the discussion above, we can see that the architecture implements software rental provided that the rented applications, e.g., 104, 105, 106, can unequivocally validate the audit trail 108. The following properties should be satisfied:

1. Holes: A rented application, e.g. word processor 104, in this present example validates that the audit trail contains all of the records that have ever been written regardless of application. If an application has previously written ten log records, then the rented application, e.g., word processor 104, would not validate the audit trail if the rented application could not locate all ten log records. We require an absence of holes, because we do not wish to permit an attacker to delete individual log records in order to destroy a record of usage.
2. Modification: An application, e.g., word processor 104, must unequivocally conclude that no unauthorized attacker modified any of WP's log records. Otherwise, for example, the attacker could modify all of the 15 minute log records to 15 second log records in order to dramatically increase the amount of time that the software may execute.
3. Current: A rented application, must be able to validate that the audit trail 108 is current. Otherwise, the audit trail 108 could potentially be old, thus hiding relatively new audit records 109, 110, 111. One would not wish, for example, an attacker to perform the backup and restore attack.

[0051] These three properties remove all incentive for an attacker to corrupt, delete, lose, or otherwise abuse an audit trail 108. If the attacker were to render the audit trail 108 invalid, then all of the rented applications 104, 105, 106 would identify the abuse and subsequently refuse rental.

[0052] In order to provide security, the architecture requires a smart card 112 that performs asymmetric cryptography. In this present example, the smart card 112 executes digital signatures. The smart card 112 contains private keying material 501 and a counter 502 (see Figure 5).

[0053] When the customer obtains the smart card 112, the smart card 112 is in a secure state. The smart card provides exactly two services that access either the private keying material or the counter: **SignAndIncrement()** and **GetCounter()**, described in pseudo program code hereinafter (note that the symbol // denotes a comment and ← denotes the assignment operator):

```
Signature SignAndIncrement(HASH h) // h is a message digest
BEGIN
     [1] Compute the message digest of h and the smart card's
counter, i.e., h' ← hash(h,counter)
     [2] Sign h' with the private keying material
     [3] Increment the smart card's counter by 1
     [4] return the digital signature computed in step [2]
END

integer GetCounter()
BEGIN
     [1] return the current value of the smart card's counter
END
```

[0054] Consider the following example trace. Suppose that the smart card's counter has an initial value of 6 and that one executes the following operations:

(i) Signature1 ← SignAndIncrement(hash(„m1"))
(ii) Signature2 ← SignAndIncrement(hash(„m2"))
(iii)int1 ← GetCounter()

[0055] The results of this example are:

- Signature1 gets the digital signature (using the smart card's private keying material) of hash(hash(„m1"),6)
- Signature2 gets the digital signature (using the smart card's private keying material) of hash(hash(„m2"),7)
- int1 gets 8

[0056] The audit trail 108 contains a list of records, where each record has four fields: nonce, string, counter, and signature. The data input into the signature is hash(hash(nonce,string),counter). Figure 6 illustrates an example audit trail with four records. In the first record, the nonce has the value 96, the string is „WP 15 minutes public key 9828a8c12a5873654bac684517d3afe3" (where 9828a8c12a5873654bac684517d3afe3 denotes the message digest of a real public key), the counter's value is 0, and the digital signature is of hash(hash(96,"WP 15 minutes public key 9828a8c12a5873654bac684517d3afe3"),0). Here, the digital signature was provided using the smart card's private keying material which corresponds to public keying material 9828a8c12a5873654bac684517d3afe3. This public keying material may be extracted from WP's keyfile.

[0057] The counter never rolls over from its highest value to zero. When the counter reaches its highest value, e.g., 2^{128}-1, the system stops.

[0058] A rented application appends a record to the audit trail by executing the *Write* routine, where the *Write* routine is embedded within the rented applications. This routine generates an audit record and then sends the audit record to the Rental Server 107. The Rental Server 107 writes the audit record into a non-volatile, stable image of the audit trail, e.g., on one or more files. The Rental Server 107 synchronizes access to the smart card 112 and the audit trail. The Rental Server 107 cannot execute in a manner that thwarts system security, i.e., the rented applications do not trust the Rental Server 107. If the Rental Server 107 were to act incorrectly, then there could potentially be a denial of service because it may be the case that the rented applications could not validate the audit trail.

[0059] The *Write* routine is provided below in pseudo program code:

```
Boolean Write(String str)
BEGIN
    [1] n ← generate a new nonce
    [2] h1 ← hash( n,str)
    [3] s ← SignAndIncrement(h1)
    // below, c is a local copy of value in the smart card
    [4] c ← GetCounter()
    // below, decrement by 1 has no affect on smart card
    [5] decrement c by 1
    [6] h2 ← hash(h1,c)
    [7] validate that s is the signature of h2 against the
        public key found in the keyfile (if the validation
        fails, then return failure immediately without
        executing any further steps; the keyfile will be
        described later in detail).

    [8] create the audit trail r ← <n,str,c,s>
    [9] append r to the audit trail
    [10] return TRUE if all of the preceding steps succeed,
         otherwise return failure
END
```

[0060] The *ValidateTrail* routine is also embedded in the rented application and should be executed periodically and is provided below in pseudo program code (assume that the system started with an initial counter value of zero):

```
Boolean ValidateTrail()
BEGIN
     [1] c ← GetCounter()
     [2] Write(c)    // use the Write routine above, exit if
                     // failure
     [3] r ← Last record in the audit trail // this is the
         record that we just wrote in step [2]
     [4] Validate the signature stored in r against the
         public key stored in the keyfile
     [5] Validate that c is the same as the counter stored
         in r
     [6] FOR i←0 UNTIL there are no more records,
         INCREMENT i by 1
          [6.1] r ← i^th record from the audit trail
          [6.2] Validate the signature stored in r against
                the public key stored in the keyfile
                // Signature validation comprises the message
                // digest recomputation
          [6.3] Validate that i is the same as the counter
stored in r
     [7] END FOR LOOP
     [8] if all of the above steps succeed, then return TRUE,
otherwise return FALSE
END
```

[0061] In steps [4] and [6.2], all of the input for the validation is from the audit record itself. By carefully analyzing all of the steps in the *Write* and *ValidateTrail* routines, it is clear that any attack that thwarts the intended use of these routines causes failure. In this case, the rented applications notice the failure and do not permit themselves to be rented.

[0062] The mechanism for recovering from a failure depends upon the vendor for each particular rented application. The vendor may, for example, issue a new keyfile upon some customers' requests. Perhaps, the most significant issue is recovery from accidental loss of the audit trail that may occur due to a disk error. In order to protect against this situation, the Rental Server 107 should be responsible for writing all records to the audit trail on behalf of all rented appli-

cations. The Rental Server should maintain a primary audit trail on the local fixed disk and at least one backup audit trail on a separate medium, e.g., a floppy or a network file server. One may, for example, provide a service that permits the Rental Server to e-mail audit trails to a secured network backup service. In this case, one can ensure privacy by permitting the rental server to encrypt the audit trail before transmission.

Software Components (see Figure 2)

[0063] We use the term challenge means to denote the portion of a rented program that implements components of this invention. The challenge means includes the implementation of the Write and ValidateTrail routines. In addition, the challenge means instructs the remainder of the program concerning how to operate depending upon the results of the Write and ValidateTrail routines. In addition, the challenge means interacts with the program's keyfile as described below.

Keyfile 404

[0064] The creation of the keyfile 404 is performed by a keyfile generator, which is a program that executes at the vendor's facility. The vendor 401 must take care to guard this program.

[0065] In use of the keyfile generator, an operator enters the following information:

Vendor name: Vendor name is the name of the vendor's company.

Vendor password: Vendor password is the password that unlocks the vendor company's private keying material. Company employees who do not know the password cannot generate keyfiles.

Customer name: The customer name is the distinguished name of a customer (defined in [2]) for whom to generate a keyfile. The name indexes into a database of public keying material.

Keyfile name: The keyfile name is the name of a new keyfile.

[0066] After obtaining this information, the keyfile generator builds the keyfile 404, containing a customer information string (CIS) described later. Portions of the keyfile 404 appear to the customer 402 as a completely random sequence of values.

[0067] Building of the keyfile 404 involves the following operations.

[0068] First, the keyfile generator creates a file and inserts the customer's public keying material into the file, along with thousands of decoy bits. In the present example, each keyfile 404 contains approximately 480,000 decoy bits. This number of bits represents a significant amount of decoy material, yet can fit into a standard e-mail message.

[0069] Each keyfile 404 stores the CIS in a different location. Additionally, each keyfile 404 has encrypted customer information embedded in it without disclosing the required encryption key. This encrypted customer information permits a vendor to easily identify the owner of a keyfile 404 in the event that the keyfile 404 appears in a public location such as a bulletin board. The keyfile generator then encrypts and re-encrypts the keyfile (or portions of the keyfile) 705 multiple times, using different algorithms. Finally, the keyfile generator signs the keyfile 404 using the vendor's private keying material by applying a digital signature algorithm.

[0070] A keyfile is said to be validated if the challenge means of the rented application (described below) can validate the vendor's signature using the public keying material stored in the challenge means' binary and access the decrypted CIS stored in the keyfile 404.

[0071] After having created the keyfile 404, the vendor's computer sends the keyfile 404 to the customer 402 by electronic mail.

[0072] The CIS is a string that contains the customer's public keying material, the customer's access rights, and possibly a rental threshold. The access rights provide information to selectively enable functionality. For example, an access right may potentially authorize the program to execute the print function. A different access right could potentially permit the application to send audio information to the customer's speakers. Presumably, highly trusted customers, or customers who pay more money obtain better access rights in their keyfiles. The keyfile 404 may have multiple uses, e.g., authorization for the application and rental information. In the following, the operation of the rental mechanism is described in more detail. This is performed when the customer initially attempts to execute the rented program.

[0073] When the challenge mechanism 202 starts the process, the challenge mechanism 202 (see Figure 7) accesses the keyfile 404 associated with the rented application and calls a signature validation function 203 in the challenge mechanism 202 to validate the vendor's signature of the keyfile 404, using the vendor's public keying material 201 that is embedded in the challenge mechanism 202. This validation of the keyfile signature ensures that an attacker can-

not modify the keyfile 404 or its digital signature without additionally modifying the challenge mechanism 202. The vendor 401 may optionally augment this protection using additional proprietary lines of defense. If the keyfile 404 has been modified, the challenge mechanism 202 hangs the rented program, or otherwise disturbs normal program execution.

[0074] Assuming the signature of the keyfile 404 is validated, the challenge mechanism 202 then parses the keyfile 404, using a proprietary, vendor-specific algorithm, to locate the customer's public keying material in the keyfile 404, and extracts the customer's public keying material.

[0075] Subsequently, whenever required by the software rental mechanism, the challenge mechanism (challenge means) executes the software rental protocol as described earlier. That is, the challenge means securely appends audit records to the audit trail as described earlier.

Nonce generator

[0076] Generation of a nonce is performed by a nonce generator included in the challenge mechanism 202. Operation of the nonce generator is as follows (see figure 3).

[0077] First, the nonce generator queries a large number of system parameters, e.g. the system time, the amount of space remaining free in the page table, the number of logical disk drives, the names of the files in the operating system's directory, etc.

[0078] Next, the nonce generator builds a random number, using a random number generator. The random number generator consists of two process threads, referred to herein as Thread 1 and Thread 2. Figure 5 shows the operation of Thread 1, which is the main thread of the random number generator.

[0079] (Box 301) Thread 1 first creates a data structure value_list, for holding a list of counter values. The list is initially empty.

[0080] (Box 302) Thread 1 sets a current counter value to zero, and sets a done_test flag to FALSE.

[0081] (Box 303) Thread 1 then forks Thread 2. Thread 2 posts an asynchronous disk access, and then sleeps until the disk access is complete. When the disk access is complete, Thread 2 sets the done_test flag to TRUE. Note that Thread 1 and Thread 2 share the done_test flag.

[0082] (Box 304) Thread 1 increments the counter value by one.

[0083] (Box 305) Thread 1 then tests whether the done_test flag is now TRUE, indicating that the disk access initiated by Thread 2 is complete. If done_test flag is FALSE, the thread returns to box 54. Thus it can be seen that, while waiting for the disk access to complete, Thread 1 continually increments the counter value.

[0084] (Box 306) When done_test flag is TRUE, Thread 1 terminates Thread 2, and saves the counter value in the first free location in value_list.

[0085] (Box 307) Thread 1 then calls a Statstest function, which estimates the degree of randomness of the counter values (or portions of counter values, e.g., low-order bits) saved in value_list. This function may use the Chi-Square Test, the Kolmogorov-Smirnov Test, or the Serial Correlation Test, which are described in [5]. The Statstest function may be optimized to ensure that complicated calculations are not repeated for each disk access. The Statstest function returns a value which indicates how many low-order bits of each saved counter value should be considered random.

[0086] (Box 308) Thread 1 compares the value returned by the Statstest function when combined with the length of the value_list with a predetermined threshold value, to determine whether enough random bits have now been generated. If not enough random bits have been generated, the process returns to box 302 above, so as to generate and save another counter value.

[0087] (Box 309) When the required number of random bits has been generated, Thread 1 extracts the specified number of low-order bits from each counter value in the value_list, and returns this sequence of bits as the output random number.

[0088] In summary, it can be seen that the random number generator exploits the unpredictability in the timing of a series of disk accesses as a source of randomness in the generation of nonces (see [4]). By forking new threads on each disk access, the random number generator also exploits unpredictabilities in the operation of the operating system's scheduler as a second source of randomness.

[0089] The analysis performed by the Statstest function permits the random number generator to self-tune for any speed processor and disk, by computing the number of low-order bits of each saved counter value to return. For example, a system with a high-variance disk access time will generate more random bits per-disk access than a system with a low-variance disk access time. For example, for a Quantum 1080s disk (6ms average write time), and a 486 66 Mhz processor, the system generates approximately 45 bits per second. Alternatively, one may hard code the number of bits per-disk access and use a de-skewing technique to ensure a good degree of randomness.

[0090] The nonce generator also queries the operating system to ensure that it posts each disk access to an actual disk. The final output nonce is formed by combining the output random number from the random number generator with the result of querying the system parameters as described above using a message digest.

[0091] The nonce generator described above works best when executing on an operating system that provides direct

access to the disk, e.g., Windows 95 or Windows NT 4.0. In such an operating system, special operating system calls available to programs executing in customer space permit a program to bypass the operating system's internal buffering mechanism and write directly to the disk. Most programs do not take advantage of these special operating system calls because they may be relatively inefficient and difficult to use. On Windows 95 and Windows NT, a program may only use these special calls if the program accesses data that is a multiple of the disk's sector size by querying the operating system.

[0092] If the operating system does not provide direct access to the disk, then the challenge mechanism 24 could still use the disk timing random number generator. However, in this case, the quality of the generated values would have a greater reliance upon unpredictabilities in the operating system's scheduler as opposed to the variance inherent to the disk access time.

[0093] The example of the invention described above assumes that the operating system permits a program to fork multiple threads within a single address space. Additionally, the example of the invention assumes that the operating system permits the threads to access synchronization variables such as semaphores. Most modern operating systems provide these services. The example of the invention uses multiple threads to implement a mechanism which quantifies each disk access time. However, if an implementation of the invention were to execute on a system that does not provide multiple threads or synchronization variables, then the nonce generator could substitute other mechanisms, e.g. querying a physical clock.

Some Possible Modifications

[0094] In order to improve performance, one may augment the system with a trusted audit trail validation service. Here, the trusted service periodically validates an audit trail and then appends a new audit record, the validator record, that securely vouches for the previous records. Example information that the validator record may contain is a digital signature of the hash of all preceding audit records in the audit trail (the digital signature uses the private key of the audit validation service). Henceforth, rented applications need not validate digital signatures of records that precede the validator record. The audit trail validation service could be implemented by a third party that is accessible via a network or e-mail connection.

[0095] In the *ValidateTrail* procedure's steps [5] and [6.3], it is possible that the counter value of the initial record is not zero. In this case, the counter value starts at *offset*, and steps [5] and [6.3] must take *offset* into account in the comparison.

[0096] Note that the vendor of the rented application trusts the smart card to avoid releasing the private keying material. Additionally, the vendor of the rented application trusts that the smart card uses its private keying material in no functions other than SignAndIncrement and GetCounter. The vendor of the rented application may wish to validate the smart card manufacturer and/or personalizer.

[0097] A vendor may create and distribute an application after a customer obtains a smart card. In this case, the vendor simply creates a software rental keyfile for the customer and sends the keyfile to the customer (possibly after receiving payment).

Universal Private Keying Material

[0098] In a variant to the example mechanism, all customers rent the software using the same public/private key pair. Here, the vendor trusts that the smart card operates correctly, and never releases the value of the private keying material outside of the smart card. As in the case of Figure 5, the smart card contains both private keying material 501 and a counter 502. Additionally, the smart card contains a unique serial number, where no two smart cards have the same serial number. Step [1] of the **SignAndIncrement** routine implemented on the smart card differs from the one described above as follows:

[1] Compute the message digest of h and the serial number and the smart card's counter, i.e.,
h' ← hash(h,serial_number,counter)

[0099] In addition to the **SignAndIncrement** and the **GetCounter** routines, the smart card additionally provides the **GetSerialNumber** routine:

```
string GetSerialNumber()
BEGIN
     [1] return the smart card's serial number .
END
```

[0100] The customer additionally sends his or her smart card's serial number. The keyfile 404 contains the following information:

- The universal public keying material potentially shared by all customers
- The unique serial number of the customer's smart card

The hash records stored in the log file, take into account the serial number. For example, the first hash record of Figure 6 has the following information for a message signature:

Signature of hash(hash(96,WP...),serial_number,0)

[0101] The *Write* routine step [6] has the following modification:

[6] h2 ← hash(h1,serial_number,c)

[0102] The *ValidateTrail* routine's steps [4] and [6.2] must use the serial number (otherwise they would always fail). We do not specify the vehicle that the *Write* and *ValidateTrail* routines use to obtain the serial number of the local smart card. One could, for example, query the smart card a single time and then store the serial number in a file in the local file system.

[0103] A software vendor could potentially provide a rented application with a complex threshold calculation. For example, the vendor may rent blocks of 1000 *units*. For each hour that the software executes between 20.00 (8:00 PM) and 6.00 (6:00 AM) the next morning, the log record defines one unit; and for each hour in a different portion of the day, the log record defines two units. So, for example, a customer could potentially use the software for 1000 nighttime hours, 500 daytime hours, or some calculated combination of nighttime and daytime hours.

[0104] Alternatively a threshold may be calculated using a boolean combination of multiple parameters. A software vendor could potentially provide a rental application with no threshold. Every month, the software vendor obtains the current value of the SUSL and gets the audit trail. The vendor then validates the audit trail and charges a fee accordingly. If the customer does not wish to continue renting the software, then the vendor asks to have the smart card returned. It is very easy to query the value of the SUSL if the customer briefly returns the smart card to the vendor. Otherwise, the vendor accesses the smart card via a network interface. First, the vendor generates a random nonce and asks the customer to use Write(nonce) to append the nonce to the audit trail. Next, the vendor executes the *ValidateTrail* routine remotely. This service requires the customer (or programs that operate on the customer's behalf) to relay the vendor's requests from the network to the smart card. The vendor then checks the validated audit trail for the nonce.

Software Copy Protection

[0105] The software rental mechanism may be augmented with an additional software copy protection mechanism. The software copy protection mechanism ensures that only the authorized customer may rent the program. The software copy protection mechanism may potentially share a keyfile with the software rental mechanism.

Protected Content

[0106] Any of the rental mechanisms described in this document may also be used for renting documents as opposed to software. The approach is to rent a Viewer program which uses a file.

[0107] The following publications are cited in this document:

[1] Hardlock API, Manual Implementation of Harlock Software Protection Systems, High-Level API Version 3 Application Programming Interface, FAST Software Security-Group, FAST Document: High-Level API, Revision: 4.00e, Status March 1, 1996

[2] A. Menezes et al, Handbook of Applied Cryptography, CRC Press, Inc. ISBN 0-8493-8523-7, S. 22 - 23, 224 - 233, 250 - 259, 308 - 311, 405 - 424, 433 - 438, 572 - 577, 1997

[3] R. Rivest, The MD5 message-digest algorithm, RFC 1321, April 1992.

[4] P. Fenstermacher et al, Cryptographic randomness from air turbulence in disk drives, Advances in Cryptology: Crypto '94, pp. 114 - 120, Springer Verlag, 1994

[5] D. Knuth, The Art of Computer Programming, Vol. 2, Seminumerical Algorithms, Addison-Wesley Publishing Co., Reading MA, 2nd Edition, 1981, pp. 38-73, ISBN 0-201-03822-6.

Claims

1. A software rental system comprising at least one rented program permitting at least one service to a customer with a customer's response means, wherein

   - said rented program has no access to a customer's private keying material,
   - using asymmetric cryptography, said customer's response means proves to the rented program, that said customer's response means has access to the customer's private keying material, and
   - said rented program does not permit said at least one service to said customer unless the proof is successful.

2. A software rental system comprising at least one rented program and private keying material such that the software rental system securely stores audit trails, wherein the correctness of said audit trails is validated using asymmetric cryptography.

3. A software rental system according to claim 2 wherein the at least one rented program has no access to the private keying material.

4. A software rental system according to claim 2 or 3 wherein the audit trails are comprising the extent to which the rented program has been rented.

5. A software rental system according to claim 1 wherein validation ensures all of the following:

   - no audit trail has been modified,
   - no audit trail has been deleted, and
   - the list of all audit trails is up to date.

6. A software rental system in accordance to one of the claims 2 to 5 in which the rented program fails to execute or executes in a limited mode if said validation fails.

7. A software rental system in accordance to claim 0.5 in which the system may rent more programs than available Secure Updatable Storage Locations.

8. A software rental system in accordance with claim 7 in which only a single Secure Updateable Storage Location is required in order to rent multiple programs.

9. A software rental system in accordance with claim 8 in which a first part of the audit trail is stored on a non-secured media and a second part of the audit trail is stored in a secure updateable storage location, wherein the first part of the audit trail comprises an amount of stored audit trails.

10. A software rental method using a software rental system in accordance with claim 9 in which a customer obtains and uses a secure updateable storage location for renting software and subsequently obtains new rented software without requiring a new secure updateable storage location or resetting an existing secure updateable storage location value.

11. A software rental method in accordance with claim 10 in which multiple rented programs are using the same rental mechanism despite the fact that the rented programs were created without knowing common private keying material.

12. A software rental system according to claim 1 with multiple rented programs.

13. A software rental system according to claim 12, wherein

- said rented programs have the same keying material, and
- customer's private keying material of said keying material is stored on a smart card.

14. A software rental system according to one of the claims 1 to 11, wherein a customer's private keying material is stored on a smart card.

15. A software rental system according to one of the claims 1 to 14, wherein

- at least one usage parameter of said at least one program is stored in an audit trail, and
- said rented program does not permit said at least one service to said customer unless said at least one usage parameter is within a predetermined threshold.

16. A software rental system according to claim 15, wherein at least two usage parameters of said at least one program are stored in said audit trail.

17. A software rental system according to claim 15 or 16, wherein said at least one usage parameter is a rental time period for renting said at least one program.

18. A software rental system according to one of the claims 15 to 17, wherein said at least one usage parameter is an amount of usages of said at least one program.

19. A software rental system according to one of the claims 15 to 18, wherein

- said at least one usage parameter is checked multiple times, and
- said rented program does not permit said at least one service to said customer unless said at least one usage parameter is in a predetermined interval each time it is checked.

20. A software rental system according to one of the claims 1 to 19, with a rental server that synchronizes access to a smart card on which a customer's private keying material is stored and/or to an audit trail in which usage parameters of said at least one program is stored.

21. A software rental system according to one of the claims 15 to 20, wherein said program stores said at least one usage parameter in said audit trail.

22. A software rental system according to claim 21, wherein said audit trail is stored securely.

23. A software rental system according to one of the claims 13 to 22, wherein the smart card performs an atomic routine that both executes a digital signature and an operation computed over a storage location that changes over time.

24. A software rental system according to one of the claims 1 to 23, wherein the program is a viewer program which uses a file.

25. A software rental system according to one of the claims 1 to 24, wherein the system includes a keyfile for holding public keying material.

26. A software rental system according to claim 25, wherein said public keying material held in said keyfile is cryptographically secured, whereby it is computationally infeasible to alter any portion of the keyfile, including said public keying material, without altering the challenge means.

27. A software rental system according to claim 26, wherein said keyfile includes information identifying said customer said program has been supplied.

28. A software rental system according to claim 27, wherein said keyfile includes decoy bits for disguising the said public keying material held therein.

29. A software rental system according to one of the claims 1 to 28, wherein the program is copy protected.

30. A method of distributing a program to a plurality of customers wherein each customer has a software rental system according to claim 1 or 2, and wherein every customer receives an identical copy of said program.

31. A method for renting software comprising at least one rented program permitting at least one service to a customer with a customer's reponse means, wherein

- said rented program has no access to a customer's private keying material,
- using asymmetric cryptography, said customer's reponse means proves to the rented program, that said customer's reponse means has access to the customer's private keying material, and
- said rented program does not permit said at least one service to said customer unless the proof is successful.

32. A method for renting software comprising at least one rented program and private keying material such that the software rental system securely stores audit trails, wherein the correctness of said audit trails is validated using asymmetric cryptography.

33. A method according to claim 32 wherein said at least one rented program has no access to the private keying material.

34. A method according to claim 32 or 33 wherein the audit trails are comprising the extent to which the rented program has been rented.

35. A method according to claim 32 wherein validation ensures all of the following:

- no audit trail has been modified,
- no audit trail has been deleted, and
- the list of all audit trails is up to date.

36. A method in accordance to one of the claims 31 to 35 in which the rented program fails to execute or executes in a limited mode if said validation fails.

37. A method in accordance one of the claims 31 to 36 in which the system may rent more programs than available Secure Updatable Storage Locations.

38. A method in accordance with claim 37 in which only a single Secure Updateable Storage Location is required in order to rent multiple programs.

39. A method in accordance with claim 38 in which a first part of the audit trail is stored on a non-secured media and a second part of the audit trail is stored in a secure updateable storage location, wherein the first part of the audit trail comprises an amount of stored audit trails.

40. A software rental method using a software rental system in accordance with claim 39 in which a customer obtains and uses a secure updateable storage location for renting software and subsequently obtains new rented software without requiring a new secure updateable storage location or resetting an existing secure updateable storage location value.

41. A method according to one of the claims 31 to 40 with multiple rented programs.

42. A method according to claim 41, wherein

- said rented programs have the same keying material, and
- customer's private keying material of said keying material is stored on a smart card.

43. A method according to claim 42, wherein a customer's private keying material is stored on a smart card.

44. A method according to one of the claims 31 to 43, wherein

- at least one usage parameter of said at least one program is stored in an audit trail, and
- said rented program does not permit said at least one service to said customer unless said at least one usage parameter is within a predetermined threshold.

45. A method according to claim 44,
wherein at least two usage parameters of said at least one program are stored in said audit trail.

46. A method according to claim 44 or 45,
wherein said at least one usage parameter is a rental time period for renting said at least one program.

47. A method according to one of the claims 31 to 46,
wherein said at least one usage parameter is an amount of usages of said at least one program.

48. A method according to one of the claims 31 to 47, wherein

- said at least one usage parameter is checked multiple times, and
- said rented program does not permit said at least one service to said customer unless said at least one usage parameter is in a predetermined interval each time it is checked.

49. A method according to one of the claims 31 to 48,
with a rental server that synchronizes access to a smart card on which a customer's private keying material is stored and/or to an audit trail in which usage parameters of said at least one program is stored.

50. A method according to one of the claims 31 to 49, wherein said program stores said at least one usage parameter in said audit trail.

51. A method according to claim 50, wherein said audit trail is stored securely.

52. A method according to one of the claims 31 to 51, wherein the smart card performs an atomic routine that both executes a digital signature and an operation computed over a storage location that changes over time.

53. A method according to one of the claims 31 to 52, wherein the program is a viewer program which uses a file.

54. A method according to one of the claims 31 to 53,
wherein the system includes a keyfile for holding public keying material.

55. A method according to claim 54,
wherein said public keying material held in said keyfile is cryptographically secured, whereby it is computationally infeasible to alter any portion of the keyfile, including the public keying material, without altering the challenge means.

56. A method according to claim 55,
wherein said keyfile includes information identifying said customer said program has been supplied.

57. A method according to claim 56,
wherein said keyfile includes decoy bits for disguising the said public keying material held therein.

58. A method according to one of the claims 31 to 57, wherein the program is copy protected.

FIG 1

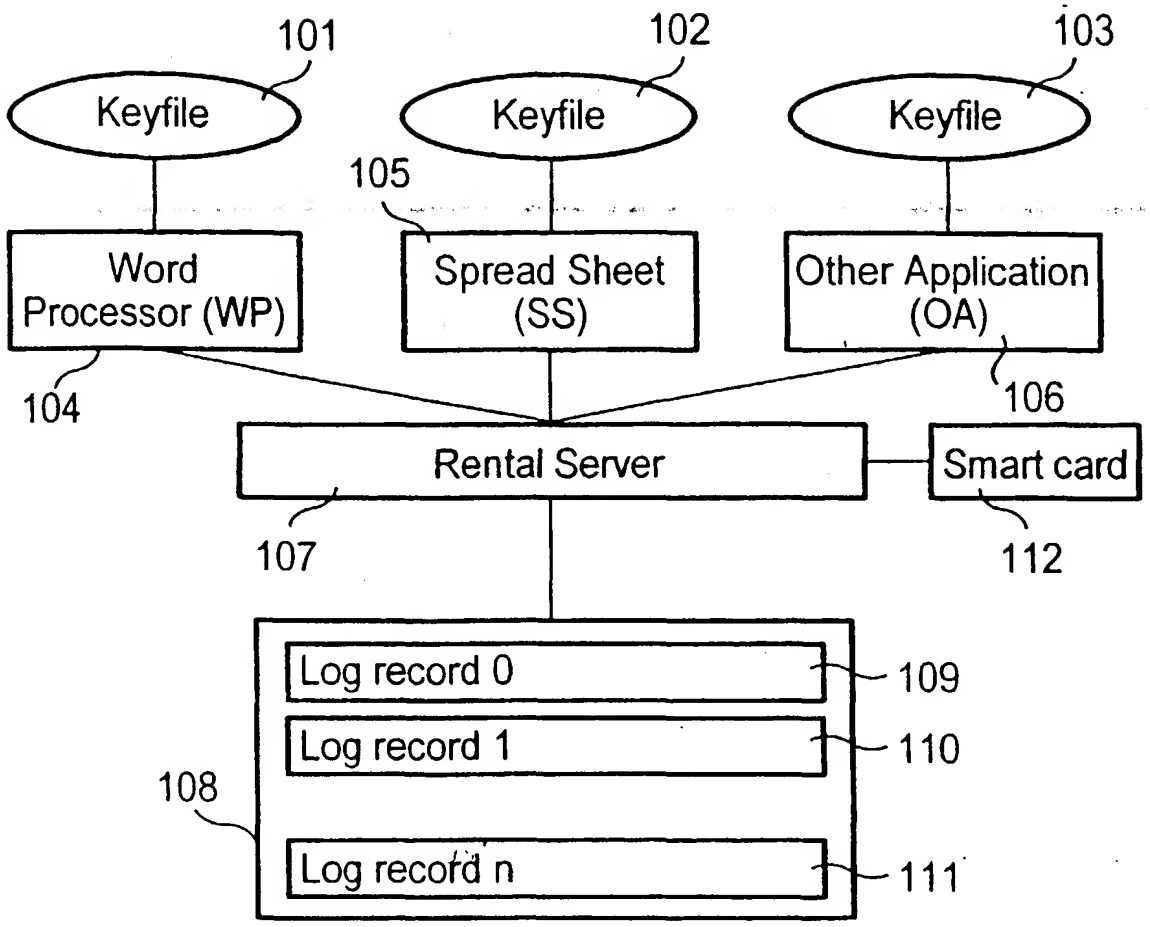


FIG 2

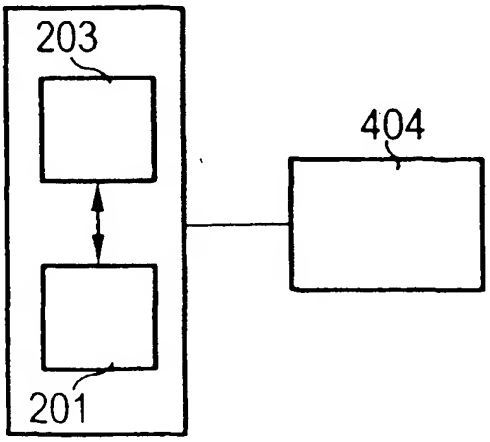


FIG 3

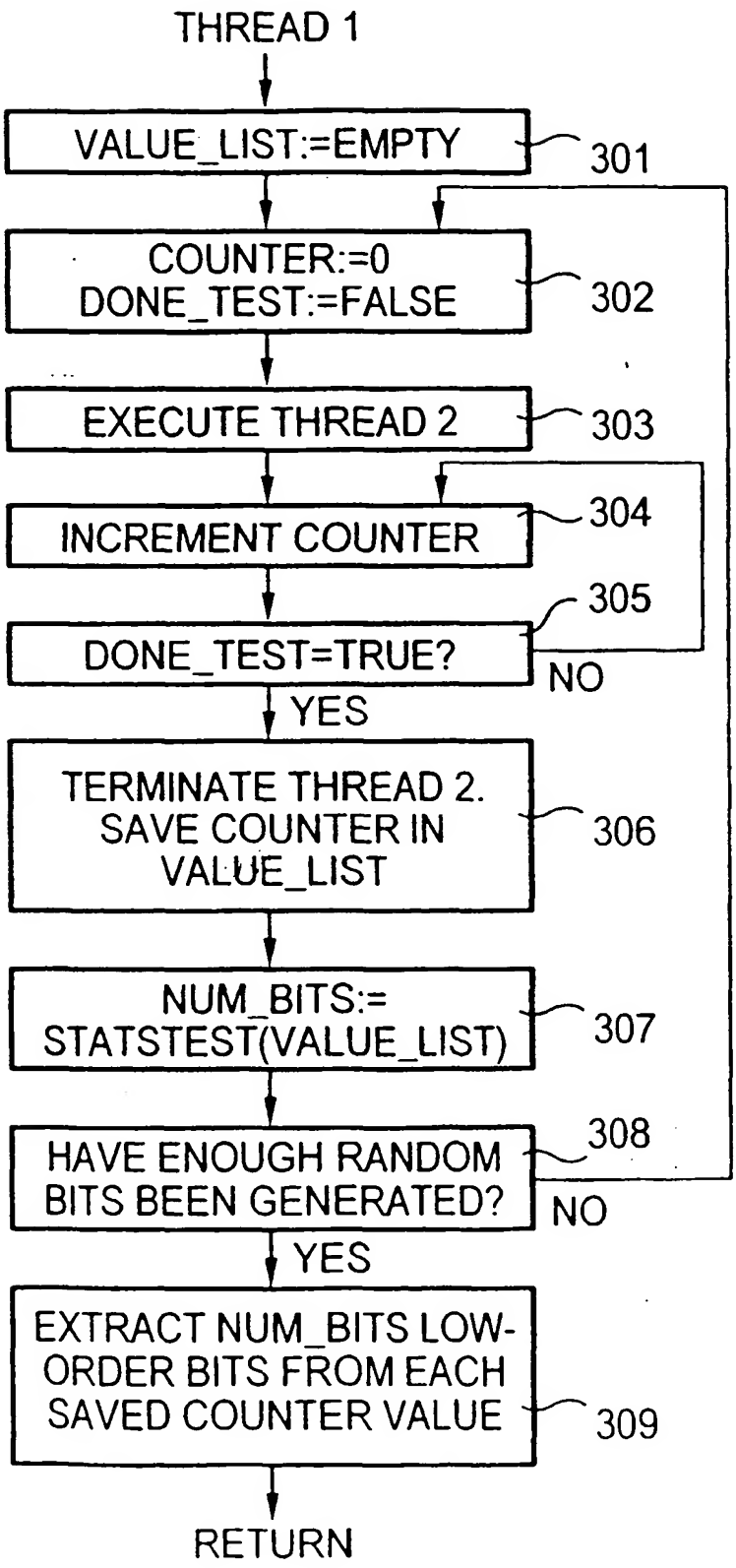


FIG 4

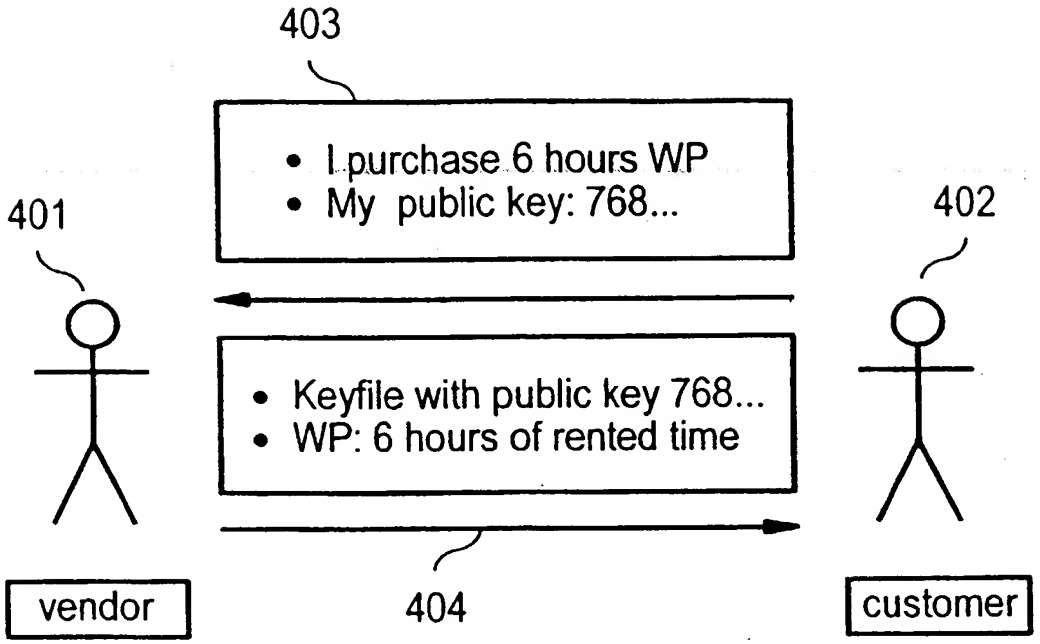


FIG 5

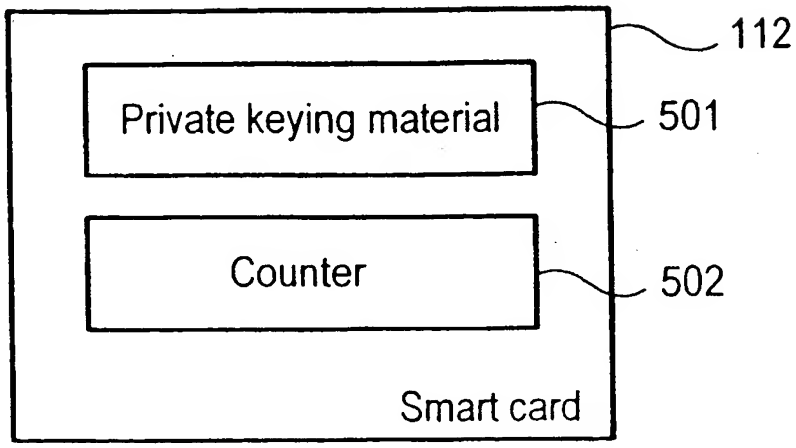
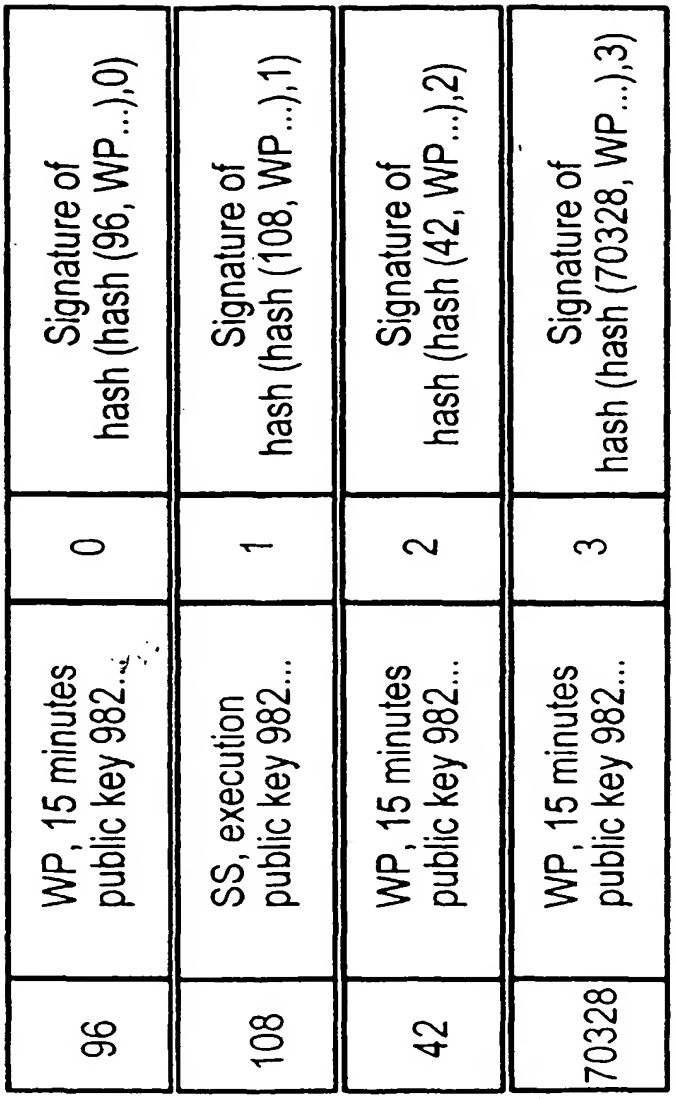


FIG 6



European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 97 11 3262

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.6)
A	US 5 023 907 A (JOHNSON HERRICK J ET AL) * abstract * * column 4, line 14 - line 40; figures *	1,2,4,5, 12,15, 17,18, 30-32, 34,35, 41,44, 46,47	G06F1/00
A	WO 88 05941 A (SOFTWARE ACTIVATION INC) * page 8, line 14 - page 10, line 2 * * page 11, line 9 - page 14, line 15 * * page 18, line 1 - page 19, line 19 * * page 24, line 11 - line 22; figures *	1,2, 30-32	
A	EP 0 325 777 A (IBM) * abstract *	2,5,32, 35	
A	WO 95 17732 A (ANANDA MOHAN) * page 5, line 3 - page 11, line 11 *	1,2,31, 32	TECHNICAL FIELDS SEARCHED (Int.Cl.6) G06F H04L

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	16 December 1997	Moens, R

CATEGORY OF CITED DOCUMENTS

X particularly relevant if taken alone
Y particularly relevant if combined with another document of the same category
A technological background
O non-written disclosure
P intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)